Module 2: The design recipe

Readings:

- HtDP, Sections 1-3
- Survival Guide on assignment style and submission
- Style Guide

We are covering the same material as the text, but using different examples and style details. Lectures do not cover all the details, so keeping up with the readings is important.

Programs as communication

Every program is an act of communication:

- between you and the computer
- between you and yourself in the future
- between you and others

Comments

Comment: information important to humans but ignored by the computer

Code: anything that is not a comment; machine-readable

```
;; Constants for calendars
(define year-days 365) ; not a leap year
```

Use one semicolon to indicate that the rest of the line is a comment.

Use two semicolons to start a line that is a comment.

Do not use DrRacket's Comment Boxes.

Goals for software design

Misconception: correctness and speed only.

Partial list of goals for programs: compatible, composable, *correct*, durable, *efficient*, *extensible*, *flexible*, maintainable, portable, *readable*, *reliable*, reusable, *scalable*, usable, and useful.

The design recipe

- A development process that leaves behind a written explanation of the development.
- Use it for every function you write from now in CS115.
- Results in
  - reliability (the function has been tested)
  - readability (comments make it understandable)
- The basic recipe will be modified as we add new ingredients.

The design recipe components

The steps will appear in the following order:

- Purpose: Describes what the function calculates. The role of the parameters in the calculation should be made clear.
- Contract: Includes the type of arguments the function consumes and the value it produces. Includes additional required conditions on the inputs if needed.
- Examples: Shows the typical use of the function. Actual number of examples depends on the problem.

More design recipe components

- Definition: Racket code for the header and body of the function. Referred to as the ”implementation”.
- Tests: A set of inputs and expected outputs used to build evidence that the function meets its requirements. Actual number of tests depends on the problem and the implementation. Choose some tests that will help you identify problems in your code (special cases, boundaries, etc.).

Using the design recipe

Exercise: Write a function that sums the squares of two numbers.

Function header:

- Determine function name: sum-of-squares
- Determine parameters: p1 and p2, the two numbers

```
(define (sum-of-squares p1 p2) ...)
```

Purpose:

- Draft a sentence to explain what the function does.
- Include a basic function call
- Use the parameter names in your explanation to show how the produced value is related to the consumed parameter values.

```
;; (sum-of-squares p1 p2) produces the sum of
;;     the squares of p1 and p2
```

Contract:

- Determine what type each parameter should be.
- Determine what type of value is produced by the function.
- Follow the format precisely:

;; fun-name: type1 type2 ... typek $\rightarrow$ type-produced

Mathematically: sum-of-squares: $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, so we write:

;; sum-of-squares: Num Num $\rightarrow$ Num

Examples:

- Choose examples that show common usage of the function.
- Choose values for parameters and determine the answer.
- Develop the examples as: fn-call $\Rightarrow$ answer
- For example, (sum-of-squares 3 4) $\Rightarrow$ 25
- Another example, (sum-of-squares 0 -2.5) $\Rightarrow$ 6.25
- Convert to Racket code: (check-expect fn-call answer)

```
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 -2.5) 6.25)
```

Function definition:

- Write the body of the function.
- Combine with the header of the function to complete the implementation.

```
(define (sum-of-squares p1 p2)
  (+ (* p1 p1) (* p2 p2)))
```

Testing:

- **Tests:** a set of inputs and expected outputs, each designed to test a specific aspect of the function.
- Here, try different combinations for p1 and p2, for example, positive and negative, integer and real, zero.

```
(check-expect (sum-of-squares -1 -2) 5)
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -10 2.5) 106.25)
```

```
;; (sum-of-squares p1 p2) produces sum of squares of p1 and p2.
;; sum-of-squares: Num Num → Num
;; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)

(define (sum-of-squares p1 p2)
    (+ (* p1 p1) (* p2 p2)))

;; Tests for sum-of-squares:
(check-expect (sum-of-squares -1 -2) 5)
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -10 2.5) 106.25)
```

More design recipe details

We develop the design recipe in a different order than it appears:

Step 1: Develop **examples** and **tests.**

Step 2: Write the **Function header.**

Step 3: Draft the **purpose.**

Step 4: Determine the **contract.**

Step 5: Complete the **function body.**

Step 6: Finalize your **tests.**

Step 7: Run the program.

Step 8: Revise as needed.

Writing a purpose statement

- Explain what the function does.
- Try to use English and math notation rather than Racket code.
- Must include how the value of the parameters influences the result.
- Should not explain all the implementation details.
- Usually includes ”produces”.

Writing a contract

Our rules will be more strict than those used in the textbook.

So far, we have seen the following types (more will be added):

- Num: any numeric value
- Int: restricted to integers
- Nat: restricted to natural numbers (integers ≥ 0)
- Any: for any Racket value (no restrictions)

Additional contract requirements

If there are important constraints on the parameters that are not fully described by its type, add an additional **requires** section to ”extend” the contract.

```
;; (my-function a b c) ...
;; my-function: Num Num Num → Num
;; requires: 0 < a < b
;;                c must be nonzero
```

More about requirements

Sometimes, restrictions are needed to avoid errors:

```
;; (sqrt-shift x c) produces the squareroot of (x - c).
;; sqrt-shift: Num Num → Num
;; requires: x >= c
(define (sqrt-shift x c)
    (sqrt (- x c)))
```

Even more about requirements

Other times, restrictions are caused by the nature of the data:

```
;; (bump-grade g inc) produces g+inc, up to a maximum of 100
;; bump-grade: Num Num → Num
;; requires: 0 <= g <= 100
;;           0 <= inc <= 10
(define (bump-grade g inc)
    (min (+ g inc) 100))
```

Writing examples

- Choose “typical” inputs for examples to show what the function does.
- Determine, by hand, what the answer should be.
- More than one example may be needed.
- Examples should be chosen **before** you write your implementation.
- Examples do not need to cover all possibilities. We have tests for that.

Writing tests

- Choose a situation to test (target a specific aspect of the problem or of the code)
- Choose specific values for the function parameters
- Determine (by hand) the expected result (the known answer)
- Compare the expected to the actual value produced by your function.
- Tests don't need to be "big". Smaller is often better, because it is easier to figure out what went wrong if they fail.

Testing builds on the examples - tests are more complete.

Warnings about testing your functions

- Never use the function to determine the expected value!
- Never cut-and-paste from interactions window to definitions window.
- Some students do not include enough tests.
- Some students have lots of tests - but they mostly cover the same situation.
- Determining the correct number of tests to use is a judgement call. You have to determine it based on the problem and the implementation.

Completing examples and tests in DrRacket

The student languages offer a convenient testing method:

```
(check-expect fun-application value-expected)
```

For example:

```
(check-expect (sum-of-squares 3 4) 25)

(check-expect (abs -5) 5)
```

Note: check-expect only works with exact numbers.

The following code will result in an error.

```
(define (circle-area r) (* pi r r))
(check-expect (circle-area 1) pi)
```

When using expected or actual values which are inexact or approximations, use check-within.

(check-within fun-application value-expected tol)

checks that abs(fun-application − value-expected) $\leq$ tol

A tolerance of .00001 should typically suffice (unless told otherwise on an assignment).

(check-within (circle-area 1) pi .00001)

(check-within (sqrt 2) 1.414 0.001)

Design recipe style guide

Note that in these slides, sections of the design recipe are often omitted or condensed because of space considerations.

Consult the course style guide on the course webpage before completing your assignments.

In particular, the style guide provides more guidance on testing different types of values.

The string data type (Str)

A **Str** is a value made up of letters, numbers, blanks, and punctuation marks, all enclosed in quotation marks.

Examples: "hat", "This is a string.", and "Module 2".

String functions:

(string-append "now" "here") ⇒ "nowhere"

(string-length "length") ⇒ 6

(substring "caterpillar" 5 9) ⇒ "pill"

(substring "cat" 0 0) ⇒ ""

(substring "nowhere" 3) ⇒ "here"

More about strings

- Strings start from position 0.
- (substring t start stop) produces a string of length stop − start, assuming that 0 <= start <= stop <= (string-length t).
- Special value: the empty string "" has length 0.
- Functions can consume and produce string values.
- See the course Web site for more Str documentation (not in the textbook).

Exchanging front and back of a string

swap-parts will exchange the front and back of a string.

The *front* is the first half of the string.

The *back* is the second half of the string.

If the length of the string is odd, we'll make the front shorter by including the middle character with the back.

```
;; (swap-parts s) produces a new string like s, with front and back
;;    parts reversed.
;; swap-parts: Str → Str
;; Examples:
(check-expect (swap-parts "angle") "glean")
(check-expect (swap-parts "potshots") "hotspots")

(define (swap-parts s) ...)
```

Subtasks: finding the midpoint, extracting the front, extracting the back.

Helper functions

A **helper function** is a function that is used in the primary function to

- generalize similar expressions,
- express repeated computations,
- perform smaller tasks required by your solution, or
- improve readability of the code

Choose meaningful function names for all functions and parameters, including helper functions (do not call it helper!).

Put helper functions before the primary function.

You do not need to include tests for helper functions on assignments.

Finding the midpoint

```
;; (mid t) produces the integer part of (string-length t)/2.
;; mid: Str → Nat
;; Examples:
(check-expect (mid "cs115") 2)
(check-expect (mid "Hi") 1)

(define (mid t) (quotient (string-length t) 2))

;; Tests for mid
(check-expect (mid "") 0)
(check-expect (mid "A") 0)
```

Extracting the front

```
;; (front-part s) produces the front part of s.
;; front-part: Str → Str
;; Examples:
(check-expect (front-part "angle") "an")
(check-expect (front-part "potshots") "pots")

(define (front-part s) (substring s 0 (mid s)))

;; Tests for front-part
(check-expect (front-part "") "")
(check-expect (front-part "Z") "")
```

Extracting the back

```
;; (back-part s) produces the back part of s.
;; back-part: Str → Str
;; Examples:
(check-expect (back-part "angle") "gle")
(check-expect (back-part "potshots") "hots")

(define (back-part s) (substring s (mid s)))

;; Tests for back-part
(check-expect (back-part "") "")
(check-expect (back-part "B") "B")
```

```
;; (swap-parts s) produces a new string like s, with front and back
;;    parts reversed.
;; swap-parts: Str → Str
;; Examples:
(check-expect (swap-parts "angle") "glean")
(check-expect (swap-parts "potshots") "hotspots")

(define (swap-parts s)
   (string-append (back-part s) (front-part s)))

;; Tests for swap-parts
(check-expect (swap-parts "") "")
(check-expect (swap-parts "Z") "Z")
```

Warnings about contracts

Our contracts are comments. They are not verified by the Racket interpreter when a function is called.

For example, if we call (swap-parts 10), we get an error, but it may not be where you expect.

```
(swap-parts 10)
⇒ (string-append (back-part 10) (front-part 10))
⇒ (string-append (substring 10 (mid 10)) (front-part 10))
⇒ (string-append (substring 10 (quotient (string-length 10)))
                 (front-part 10))
⇒ ERROR
```

More about contracts

The error occurs when we cannot simplify our expression further, not simply because the contract was violated.

In CS 115, you should assume data provided in a function call wlll satisfy the contract unless you are told otherwise.

Dynamic typing vs Static typing

Some programming languages (like Java and C) require that the type of each parameter is specified when the function is defined. This is called *static typing*.

Racket does not require us to specify types ahead of time. This is called *dynamic typing*. Functions can be called with values of any type. Errors only arise when we try to use the values in ways they cannot be used (for example, when we try to take the length of an integer).

Computing a phone bill

cell-bill consumes the numbers of daytime and evening minutes used and produces the total charge for minutes used.

Details of the phone plan:

- 100 free daytime minutes
- 200 free evening minutes
- 1 dollar per minute charge for each additional daytime minute
- 50 cent per minute charge for each additional evening minute

How can we remember the meaning of each number?

Using constants

```
;; constants for phone plan
;; Free limits
(define day-free 100)
(define eve-free 200)
;; Rates per minute
(define day-rate 1)
(define eve-rate .5)
```

Use constants for readability and flexibility.

Put them before helper functions and primary functions.

```
;; (cell-bill day eve) produces cell phone bill for day daytime
;;    minutes and eve evening minutes used.
;; cell-bill: Nat Nat → Num
;; Examples:
(check-expect (cell-bill 101 0) 1)
(check-expect (cell-bill 99 0) 0)
(check-expect (cell-bill 0 199) 0)
(check-expect (cell-bill 0 202) 1)
(check-expect (cell-bill 150 300) 100)

(define (cell-bill day eve) ...)
```

```
;; (charges-for minutes freelimit rate) produces charges for minutes,
;; given the rate per minute past the freelimit.
;; charges-for: Nat Nat Num → Num
;; requires: rate >= 0
;; Examples:
(check-expect (charges-for 101 100 5) 5)
(check-expect (charges-for 99 100 34) 0)

(define (charges-for minutes freelimit rate)
  (max 0 (* (- minutes freelimit) rate)))

;; Tests for charges-for:
(check-expect (charges-for 100 100 5) 0)
```

Completing cell-bill

```
(define (cell-bill day eve)
  (+
   (charges-for day day-free day-rate)
   (charges-for eve eve-free eve-rate)))

;; Tests for cell-bill
(check-expect (cell-bill 100 0) 0)
(check-expect (cell-bill 0 200) 0)
(check-expect (cell-bill 50 175) 0)
(check-expect (cell-bill 100 200) 0)
```

Goals of this module

You should be comfortable with these terms: comment, code, contract, requirements, purpose, examples, definition, function header, body, tests, helper function.

You should know the types that are allowed in contracts so far.

You should understand the reasons for each of the components of the design recipe, the order in which they appear, and the order in which they should be created.

You should know when to use check-expect and when to use check-within.

You should start to use the design recipe and appropriate coding style for all Racket programs you write.

You should look for opportunities to use helper functions and constants to structure your programs, and gradually learn when and where they are appropriate.

You should be able to use strings in labs, assignments, and exams.